

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Domain-Specific Language Development
for Microfluidic Biochips**

Shutao Shen

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Domain-Specific Language Development
for Microfluidic Biochips**

**Entwicklung der domänenspezifischen
Sprache für Mikrofluidische Biochips**

Author:	Shutao Shen
Supervisor:	Prof.Dr.-ing Ulf Schlichtmann
Advisor:	Dr.-Ing. Tseng, Tsun-Ming
Submission Date:	15.06.2020

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.06.2020

Shutao Shen

Acknowledgments

First of all, I would like to show my respect to Prof. Dr.-Ing. Ulf Schlichtmann, head of chair electronic design automation. I'm very grateful to Dr.-Ing. Tsun-Ming Tseng, researcher and teaching assistant at the chair of electronic design automation, for providing me the interesting topic and careful guidance of my graduation. I would also like to thank to my friends, Yushen Zhang, Qingyu Li, and Yi He, for their useful help and advice.

I would also like to extend my thanks to some friends, like Junbo Xu and Hanyu Gao, who give me continuous encouragement in the hardest moment to finish this project and thesis.

Abstract

Microfluidic biochips have been gaining great achievements in academia and industry in recent years. Some algorithms are capable of the automation design of these devices. Several automation tools based on those algorithms have been published, yet there have been no standard languages for the data description and information exchange. This thesis proposes to specify a domain-specific language UMF in this field, which supports Columbus and is compatible with other tools. UMF is an extensible, object-oriented, Data Description & Schema Language. It will enable researchers to describe and validate their input and output formats quickly. The implementation of compiler and IDE will also be included in this thesis.

Contents

Acknowledgments	iii
Abstract	iv
1. Introduction	1
2. Background	3
2.1. Analysis of Special Needs	3
2.2. Current Situation	4
2.3. Limitation	4
2.4. Current Attempts	5
3. Language Design	6
3.1. Syntax and Grammar Notation	6
3.2. Language Paradigm	6
3.2.1. Object-oriented: Object-oriented Data Description	7
3.2.2. Block-structured: Package Model	7
3.3. Grammar and Semantics	8
3.3.1. Lexical grammar and syntactic grammar	8
3.3.2. JSON Format	8
3.3.3. Reserved properties	9
3.3.4. Semantics	9
3.4. Language Features	10
3.4.1. Language Structure	10
3.4.2. Control Segment (CS)	11
3.4.3. Schema Segment (SS)	14
3.4.4. Instance Segment (IS)	15
3.4.5. Package Encapsulation	15
3.4.6. Package-chain and Chained-dependencies	15
3.4.7. UMF Path	16
3.4.8. Reference and Override	17
3.4.9. Reusability (Inheritance and Composition)	17

3.5. Language Convention	18
3.5.1. Full Name and Namespace	18
3.5.2. Access Permission	18
3.6. Schema Sub-language	19
3.6.1. JSON Schema	19
3.6.2. OSON	20
3.7. Conclusion	29
4. Implementation of Compiler	30
4.1. Compiler: UMF Core	30
4.1.1. Compile Schema Segment	30
4.1.2. Compile Instance Segment	31
4.2. Automatic Lexical Analysis and Syntactic Analysis	31
4.3. Automatic Code Generation	32
4.4. Compiler: OSON	32
4.4.1. Lexical Analysis and Syntactic Analysis	33
4.4.2. Code Generation	33
4.5. Compiler: OSONA	34
4.5.1. Lexical Analysis and Syntactic Analysis	34
4.5.2. Code Generation	35
4.6. Conclusion	35
5. Implementation of IDE	36
5.1. Dynamic Packages Loading	36
5.2. Dynamic Syntax Generation and Validation	36
5.3. Debounce and Stable Strategy	37
5.4. Virtual File System in Browser	38
5.5. Auto-complement	38
5.6. Foldable JSON View	38
5.7. Monaco Package	39
5.8. Conclusion	39
6. Applications	41
6.1. Integrating UMF into "Cloud-Columba"	41
6.1.1. The Class Diagram of Columba S	41
6.1.2. Build Package "ColumbaS"	42
6.1.3. JMESPath converter	43
6.1.4. Current View	43

Contents

6.2. "Cloud-Columba" with "Parchmint" Benchmark	44
6.2.1. Conversion	44
6.2.2. Converter	46
6.3. Workflow Platform "Cloud-Columba"	46
7. Conclusion	48
List of Figures	49
List of Tables	50
Bibliography	51
A. Appendix	53
A.1. OSON Grammar	53
A.2. OSONA Grammar	56

1. Introduction

Microfluidic biochips have been gaining great achievements in academia and industry in recent years. The traditional manual design of microfluidic biochips takes much time and effort, and the whole process is error-prone. The automated design of microfluidic biochips is one of the popular technologies to reduce the investment of the design of microfluidic biochips. "Columba S", developed by Dr.-Ing. Tsun-Ming Tseng, is a scalable co-layout design automation tool for microfluidic large-scale integration with the plain-text input of rules and script[1].

```
module:                                netlist:                                conflict:
M1 d r 1 1 0 7600 3800 50              i_anti M1 1                             fin
M2 d r 1 1 0 7600 3800 50              i_anti M2 1                             parallel:
RC1 d c 0 1 0 0 3000 1200 100          i_prob M1 1                             2 M1 M2
RC2 d c 0 1 0 0 3000 1200 100          i_prob M2 1                             2 RC1 RC2
i_anti p 1500 1500                     i_kin M1 1                               fin
i_prob p 1500 1500                     i_kin M2 1
i_kin p 1500 1500                       M1 o_wf 0.5                             group:
o_wf p 1500 1500                       M1 o_wb 0.5                             fin
o_wb p 1500 1500                       M1 RC1 2
o_wf2 p 1500 1500                      M2 o_wf 0.5                             testing_module:
o_wb2 p 1500 1500                      M2 o_wb 0.5                             OFF
fin                                       M2 RC2 2                                 fin
                                          RC1 o_wf2 2
                                          RC1 o_wb2 2
                                          RC2 o_wf2 2
                                          RC2 o_wb2 2
                                          fin
```

Figure 1.1.: The Plain-text Input

Other programs, like validation-, simulation-, visualization-, benchmark-tools, are now in the process around this automated design software. A flow-path validation tool: "VOM", is one of the best validation and optimization tools to validate the output of "Columba S"[2]. A visualization tool based on VOM, provides the visualization of fluid behavior[3]. A benchmark tool, "Parchmint" provides the generation of benchmark test cases and test-based quick validations of automated design software[4]. Besides, an interactive website "Cloud-Columba" has been published to provide users with an interface to use the "Columba S"[5]. At this point, the related ecology system of the Microfluidic biochips' automated design has begun to take shape.

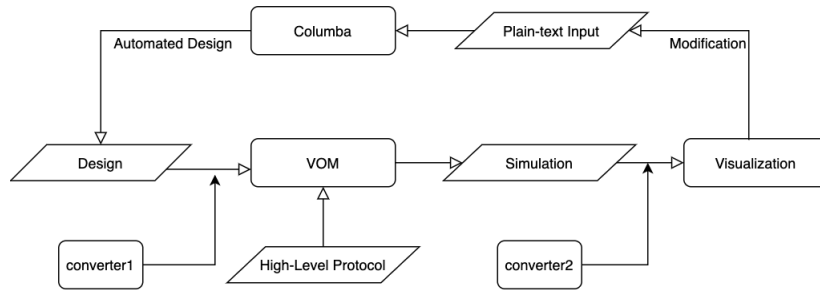


Figure 1.2.: Programs Topology

More and more tools have been created and published, and the communication of those tools is inevitable because they depend highly on each other. Until now, most tools have their individual input format, and some are plain text or script. It is challenging for other tool providers to use and integrate them. The parsing of input and generating the exact format of output are tough and tricky works.

Solving this problem should be a standard language that lets those tools easy to exchange information and convert format. In general, the primary purpose of this bachelor thesis is to implement a universal standard language of the automated design of the microfluidic biochips. In the following, I would like to introduce the background of the current situation and limitation, then describe the central concept of this language and the language features. Subsequently, the implementation of the compiler and web-based IDE (Integrated Development Environment) will be demonstrated and analyzed. At last, some applications of the language will be shown, e.g., integrating the language into "Cloud-Columba", closing the gap of "Parchmint" with "Cloud-Columba", and creating the automated workflow platform "Cloud-Columba".

2. Background

The goal is to design a domain-specific language of the automated design of microfluidic biochips. A domain-specific language (DSL) is based on the relevant concepts and features of that domain[6]. The aimed users of the language are the researchers and the tool developers in this field. Non-programmers can quickly learn the language and use it to describe things briefly and precisely. So there are the following demands and limitations that should be considered.

2.1. Analysis of Special Needs

The language should work fine in the field of the automated design of microfluidic biochips. The data in this field can be from low layer to high layer with different levels of abstraction, so the language must be a cross-layer language that can describe both the low layer metadata and the high layer information. A hierarchical language is the right solution.

The data description with programming languages is not a good idea, and it can lead to the great difficulty of the exchange process. So only the Data Description Language (DDL) and markup language will be considered. The design process of a such language is not the same as a programming language.

The language can be easily parsed and generated with the dedicated compiler and also the existed library, so it should have the general format, like JSON, YAML, XML, etc. XML file is always too big[7]. YAML's indents is a problem by deep data structures[8]. JSON, as a lightweight data-interchange format[9], can be easily integrated into multiple programming languages, and can easily read and write both by humans and machines. "JSON's structures look like conventional programming language structures. No restructuring is necessary"[9]. With this feature, JSON becomes a format of cross-language information exchange, and it almost becomes the standard of web transmission type today. Thus, the new language will be based on JSON.

The electronic components or modules are always standardized, so the encapsulation and the reusability must be two of the essential features in the language. It can

release the repetitive work of the description of the same objects in different projects. Thus, the object-oriented description is a suitable model.

Since there are until now some individual languages created by some tools developers, like "Parchmint", the language must have excellent compatibility, which allows other tool developers to close the gap with few efforts and changes. So a common schema language, like JSON Schema, should be compatibly supported.

The language must provide a high-frequency update environment for the users, allowing them to update their descriptions of data with high-frequency continuous integration because most of the projects in this field are now in fast update iteration phases. Also, the fast update means the extensibility of the language should be a particular feature. In order to achieve this requirement, "package-mode" will be applied in the language.

The "Cloud-Columba" now needs two files as the input, namely "rules" and "script". It is not suitable for a website to transfer two files instead of encapsulating them into one file. So merge the rules and script into one file should be a particular need.

2.2. Current Situation

The language is primarily aiming at the "Cloud-Columba" and some internal projects, and then the tools and programs of other researchers, so it is better to clarify the current situations of internal projects.

Currently, "Cloud-Columba" queries the plain-text script as input and outputs the meta CAD data. The "VOM" project can not directly read the output CAD data from "Cloud-Columba" as its input. The output data must be converted. The output of "VOM" can not directly be applied as the input of the visualization tool. There should also be another converter to convert the data. So, the exchange of the data inside the internal platform is a big problem. The converters make the system's efficiency low, and the individual separate data format slows down the whole development procedure. Now a unified language is necessary.

2.3. Limitation

The language is a domain-specific language, which can be regarded as a combination of Data Description Language (DDL), Schema Language (SL), and Document Markup

Language (DML). However, it is not a Turing Completeness Language, which means this language can not apply the useful statements, like the flow control statement, the loop statement, and the calculation statement, to increase the expression power. Therefore the static configuration and syntactic sugar will be applied to make the writing process more efficient.

A regular format of DSL, like OpenAPI, is not enough for the field. The tools in this area has its own scenarios. They can have enormous differences by data structure. A more general case should be considered, rather than only focusing on one specific tool. Hence, the language should contain all of the data from different tools separately.

The whole language must be finished inside JSON format, which means all of the language features, also include the input and the compiled output of the language, must be implemented inside the JSON closure.

2.4. Current Attempts

There are currently some attempts, like "Parchmint". It provides a standard for their own input and output with the JSON format and a JSON Schema to validate its JSON. However, it is not a standard language in general. Directly using "JSON Schema" is not suitable for the field automated design of microfluidic biochips because it does not contain any instance information. Since the "JSON Schema" is a schema language that builds the rules from top to bottom with macro-mode, it is suitable for mature projects which are rarely modified. Another disadvantage is that the data and rules are separate, which means multiple files of data and rules, together with the connection information to describe the relationship between them, are required. Consequently, the micro-mode with extensible packages that combine the schema and the data will be the new attempt.

3. Language Design

To achieve the goals that we describe in chapter 2, a new domain-specific language is proposed in this thesis: "Unified Microfluidic Language (UMF)", which is a corresponding Data Description Language (DDL) or Markup Language with the format JSON.

3.1. Syntax and Grammar Notation

In this article, there are three grammar description methods, namely "McKeeman Form (MF)", "Backus-Naur Form (BNF)", and "Bison grammar rule (BGR)".

"Backus-Naur form or Backus normal form (BNF) is a notation technique for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols"[10].

"McKeeman Form is a notation for expressing grammars. It was proposed by Bill McKeeman of Dartmouth College. It is a simplified Backus-Naur Form with significant whitespace and minimal use of metacharacters"[11].

Bison grammar rule (BGR) is the modified BNF or MF using in Bison to generate a state machine, so it is a state-machine-oriented grammar.

Those three grammars are variants of each other, so they are similar in form. All of them are used to reduce redundancy, and the differences between them are omitted in this article.

3.2. Language Paradigm

For programming languages, there are several language design paradigms, such as Object-oriented Programming (OOP), Procedure-oriented Programming, Functional Programming (FP), etc. The paradigms have the main influence or decision for the

execution model of the language. So we use similar terminologies in the language field Data Description Language (DDL) or markup language.

3.2.1. Object-oriented: Object-oriented Data Description

OOP has three characteristic features: Encapsulation, Inheritance and Polymorphism[12]. Similar to the OOP, language Paradigm "Object-oriented Data Description (OODD)" has the same features. UMF implements encapsulation with a concept of "package", inheritance with the syntax sugar "\$use" and "package-chain". Since UMF is DDL, there are no methods inside UMF, polymorphism is no sense. Those technologies will be explained later in this chapter. For example in Figure 3.1, Columba S is an aggregation of Script and Rules and Script is built with the old Columba package.

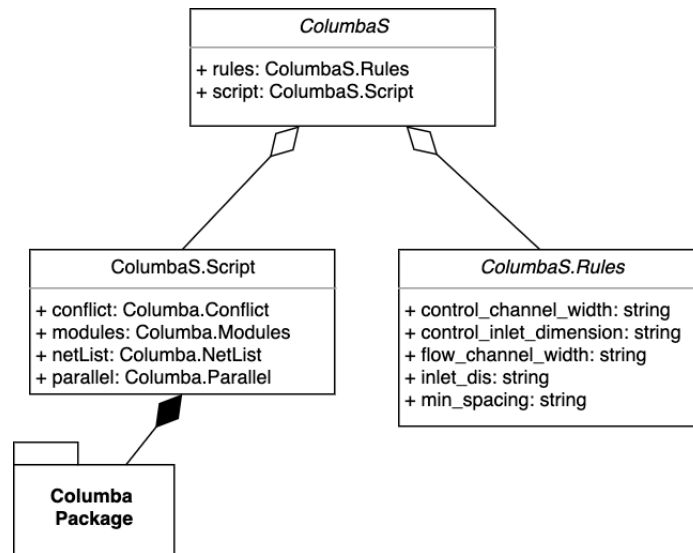


Figure 3.1.: The Class Diagram of Columba S

3.2.2. Block-structured: Package Model

There are mainly two different paradigms to achieve expansibility in DDL.

The first paradigm is "Block-structured". Every time the project data need some update, it is not necessary to update the main language. Instead, the package (block) of the project will be updated. We denoted this paradigm here as the "Package Model". For example, the language "LATEX" can dynamically add some packages. The package providers just need to change their own packages if they have some updates, and the

LATEX itself doesn't need to be changed.

The other paradigm is "Declarative Modeling". The paradigm has the description of the whole system. So there are continuous updates in the main language, and every formal update of some data will be integrated into the main language. We denoted this model as the "Integration Model". This model is specifically suitable for the area or tools with a stable information set. A very famous example is the language "OpenAPI," which "defines a standard, programming language-agnostic interface description for REST APIs"[13].

UMF will use the package model because most of our projects are still developing.

3.3. Grammar and Semantics

3.3.1. Lexical grammar and syntactic grammar

"The lexical grammar defines the set of rules for forming input elements which are the starting point of the syntactic grammar. The syntactic grammar defines the set of rules that describe how sequences of input elements can form syntactically correct programs"[14].

UMF is a valid JSON format, so the basic lexical grammar should be the same as JSON. The syntax grammar of UMF should be a subset of JSON, which means UMF's syntax is the JSON's syntax with some constraints. So the JSON parser can parse the UMF data successfully, but in other direction failed. For example, UMF's sub-language OSON redefines the "string" of JSON (see Section 3.6.2), so some words like "alice@" is not a valid string in UMF's \$schema field, but valid in JSON.

3.3.2. JSON Format

A valid JSON format is one of the following elements: object, array, string, number, true, false, and null. "A JSON object is an unordered collection of key/value pairs"[9]. It can be abstracted as a data structure: Hashtables. "The JSON array is an ordered collection of values separated by commas"[9]. It can be abstracted as "List". A full grammar definition of JSON with MF can be find in [9]. An example of JSON is shown here:

```
{
  "name": "Alice",
  "age": 18
```


}

3.3.3. Reserved properties

There are reserved properties in UMF, they are listed here:

"\$schema": The rules that will be used to check the fields.
"\$package": The rules and data that are imported from other UMF packages.
"\$name": The official package name with namespace
"\$schemaType": The type of schema, can be "JSON Schema" or "OSON".
"\$skipValidation": Skip the validation of some fields.
"\$umf.ref": The UMF style reference.
"\$apply": Auto-apply the "\$package" to "\$schema".
"\$use": Import other packages.
"\$packageName": The package name to be imported.
"\$url": The package URL to be imported.

3.3.4. Semantics

Only to discuss the syntax without semantic is meaningless in UMF, because UMF is defined over JSON, and the meaning of UMF is the defined semantics. Since UMF is a valid JSON, the syntax definition with common grammar languages, like BNF, MF, will be too complicated. JSON Schema describes the structure of JSON data, so it defines the UMF syntax grammar better. A JSON Schema definition of the core UMF syntax is here:

```
{
  "type": "object",
  "properties": {
    "$name": {"type": "string"},
    "$use": {
      "type": "array",
      "items": {"anyOf": [
        {"$ref": "#/definitions/packageUse"},
        {"type": "string"}]}
    },
    "$schemaType": {
      "type": "string",
      "enum": ["OSON", "JSON Schema"]
    }
  },
}
```

```
"$package": {
  "type": "object",
  "additionalProperties": {"$ref": "#/definitions/UMF"}
},
"$skipValidation": {
  "type": "array",
  "items": {
    "anyOf": [
      {"type": "array", "items": {"anyOf": [{"type": "string"}]}},
      {"type": "boolean"}
    ]
  }
},
"$schema": {
  "anyOf": [
    {"$ref": "#/definitions/JSONSchema"},
    {"$ref": "#/definitions/OSON"}
  ]
}
}
```

The entire syntax JSON is "#/definitions/UMF", which means the definition of "\$package" is recursive. "#/definitions/JSONSchema" and "#/definitions/OSON" are two sub-languages which will be expressed in section 3.6. No properties are required in the syntax, so all properties are optional.

3.4. Language Features

3.4.1. Language Structure

The language UMF can be regarded as a set of packages. The package is the complete independent unit in UMF. Inside every package, there are three segments: Control Segment (CS), Schema Segment (SS), and Instance Segment (IS). The segments do not have exact boundary, which means UMF allows the intersection of segments. Since JSON is equivalent to a Hashmap data structure, the fields of those three segments must be in the top layer (depth=1) of the JSON format. The following is a valid UMF file:

```
{
  // Control Segment
  "$name": "columba.modules.rings",
```

```
"$use": [ "example.package" ],

// Schema Segment
"$schemaType": "OSON",
"$schema": [
  {
    "name": "R_3_0@String='R_3_0'",
    "cellTrap?": "true@Boolean=true",
    "description?": "@String='this is a Ring'",
    "height": "150@Integer=150",
    "length": "4800@Integer=4800",
    "parallel?": "true@Boolean=true",
    "pump?": "true@Boolean=true",
    "sieveValve?": "false@Boolean=false",
    "width": "2400@Integer=2400"
  },
  "array"
],

// Instance Segment
"columba.modules.rings": [{
  "name": "thisName",
  "description": "this is a ring",
  "length": 4800,
  "width": 2400,
  "height": 150
}]
}
```

UMF makes a combination of the schema part and the instance part, which means both parts are inside a single file. Thus, no extra connection information is required to describe the relationship.

3.4.2. Control Segment (CS)

\$name

Every package has a package name. This field can be omitted if the filename is equal to the package name plus ".json" suffix.

\$package

\$package field contains the data from external packages. This field can be imported from other packages or written manually. The whole field is equivalent to an object, which means the keys are the package names, while the values are the package contents. Multiple packages can be integrated here with different keys. An example of \$package field in "columbaS.script" is in Figure 3.2:

```
    "$package": {  
      "columba.conflict": {  
        "$name": "columba.conflict",  
        "$schema": { ...  
      }  
    },  
    "columba.modules": {  
      "$name": "columba.modules",  
      "$schema": { ...  
    },  
    "$package": { ...  
  },  
  "columba.netList": { ...  
},  
"columba.parallel": { ...  
}  
}
```

Figure 3.2.: \$Package Field

\$use

"\$use" is a syntactic sugar in UMF, which means UMF is also a full language without \$use, but with \$use, the user can read and write the code more efficiently. The value of \$use should be an array. The elements of the array can be one of the following elements:

- string: the package name
- object: { "\$packageName": "Name string", "\$url": "URL string" }

The \$use field will be handled by the compiler, so after compiling, this field will be lost. The process of "desugaring \$use" will be four parts:

1. Downloading the metadata, if the data is not local.
2. Completing the "\$schema" field if the "\$schema" field does not exist. The auto-completion will apply the code template:

```
"$schema":{"$apply":true}
```

3. Translating the "\$package" part: Compile the package metadata firstly, then write the compiled output into the "\$package" field with the package name as the key. The recursive invoking process of "desugaring \$use" and "compiling package" will terminate if the package does not have a "\$use" field.
4. Translating the "\$schema" part: Copy the "\$package/packageName/\$schema" to "\$schema" field, if "\$apply" is true or contains the corresponding package name, and then merge them with the compiled original "\$schema" part as the new schema.

A simple "\$use" example is here:

```
//source code
  "$use":["columba.script", "columba.rules"]
}
//after compiling:
{
  "$package":{
    "columba.script":{
      "$name": "columba.script",
      "$schema": {...},
      "someScript": {...},
      ...
    },
    "columba.rules":{
      "$name": "columba.rules",
      "$schema": {...},
      "someRules": {...},
      ...
    }
  },
  "$schema":{
    "type": "object",
    "properties":{
      "columba.script":{...},
      "columba.rules":{...}
    }
  }
}
```

3.4.3. Schema Segment (SS)

\$schema

The `$schema` field only contains the schema that the user writes during the input, but after compiling, the `$schema` field will also include the schemas imported from `$use`. The field should be only one JSON Schema that corresponds to the whole UMF data. The keys of JSON Schema's properties are instance names, while the values are the corresponding schemas. If there is a top-level instance field of which the key-name is in the `$schema`'s properties, the corresponding schema will be used to validate this instance field. "`$schema`" also accepts OSON schema. The relationship between OSON and JSON Schema will be discussed in chapter 3 Section 3.6. An example of the `$schema` field:

```
{//source code
  "$use": [ "columbaS" ],
  "$schemaType": "OSON",
  "$schema":{
    "someInstance":"str@String"
  },
  "someInstance":123
}
{//after compiling
  "$schema": {
    "type": "object",
    "properties": {
      "someInstance": {
        "type": "string"
      },
      "columbaS": {
        "type": "object",
        "properties": {...}
      }
    }
  },
  "required": [ "someInstance" ]
},
"someInstance": 123 // throws type error
}
```

\$schemaType

\$schemaType can be either "OSON" or "JSON Schema". OSON will be compiled to JSON Schema before the validation of instances or dumping to the \$package field.

\$skipValidation

\$skipValidation can be used to skip the validation of some top-level instances. It is useful for the name conflicts inside a package.

3.4.4. Instance Segment (IS)

IS contains multiple instances in the top-level of UMF. The name of the instances must have no conflicts with the reserved words and can be a full name with namespace (see Section 3.5.1). Standard JSON members are required. In the following example, there are only two instances: "exampleNS.instance1" and "instance2".

```
{ // top-level
  "exampleNS.instance1": 123
  "instance2": { // second-level
    "attribute": "this is not an instance."
  }
}
```

3.4.5. Package Encapsulation

Every package consists of CS, SS, and IS, so it contains both schema and data. After giving a name to the package, it is encapsulated. Other users can easily use the package with "\$use" or "\$package". This is the same way how we build a package in some OOP languages, like Java or Javascript.

3.4.6. Package-chain and Chained-dependencies

Like the prototype-chain in Javascript, UMF has the concept "package-chain". A package can have a deep dependency tree[15] because of the recursive invoking of the "desugaring \$use" and the "compile" methods. The "n-depth" package can only have direct access to the instances and schemas in "(n+1)-depth" package. The package-chain will help if the instances or the schemas in deeper packages is needed.

The structure of the "\$package" can be regarded as a hashtable. All of the dependent package data is inside the "\$package" field. With the UMF Path (see subsection 3.4.7)

"#/\$package/packageName/\$package"

the cross-layer package can be accessed. An example with the reference (see subsection 3.4.8) is shown here:

```
{
  "$package": {
    "columba.modules": {
      "$name": "columba.modules",
      "$schema": {...},
      "$package": {
        "columba.modules.inlets": {
          "$name": "columba.modules.inlets",
          "bigInlet": {...}
        }
      }
    }
  },
  "package-chain": {
    "$umf.ref": "#/$package/columba.modules/$package/columba.
      -modules.inlets/bigInlet"
  }
}
```

3.4.7. UMF Path

There are two path types in UMF: "Absolute Path" and "Relative Path".

"Absolute Path" is the same definition as "JavaScript Object Notation (JSON) Pointer"[16]. Both "JSON String Representation (JSR)" and "URI Fragment Identifier Representation (UFIR)" are supported in UMF. Absolute Path starts with "/" or "#" as the root of JSON and uses "/" for the next level attributes. Array elements can be selected by "/0", where the number 0 is the index of the array element. A simple example from RFC6901 is shown in Table 3.1

"Relative Path" is a package-based path system. It starts with a package name and then follows the same rules as Absolute Path. For example, the following two paths are equivalent:

	JSR	UFIR	Result
{			
"foo": ["bar", "baz"],	"/foo"	"#/foo"	["bar", "baz"]
"a/b": 1	"/foo/0"	"#/foo/0"	"bar"
}	"/a 1b"	"#/a 1b"	1

Table 3.1.: An example from RFC6901

```
"#/$package/myPackage/myData"
"myPackage/myData"
```

3.4.8. Reference and Override

"Reference" and "Override" are syntactic sugars in UMF. They have the similar structure:

```
{
  "$umf.ref": "refData",
  // override properties if it is a override
  "someOverrideProperties": "newValue"
}
```

The reference is a deep-copy of the referred object, while the override is a partial deep-copy of the referred object. An override can be decomposed into two parts: dereference process and modification process. The dereference process copies the object with the UMF Path, and the modification process changes the property values.

Different from JSON Schema, the UMF style reference does not support ID-Reference, a reference that uses "id" flag to resolve the data in JSON Schema[17], because the ID-Reference will increase the chaos degree. The UMF style reference can reduce the repetitive descriptions of instances and schema segments. The compiler will handle the reference and the override after the "desugaring \$use" process.

The JSON Schema has only rules, so it does not have the concept "Override". However, in UMF, there are instances. "Override" is a crucial feature that can lead to "composition-mode" inheritance.

3.4.9. Reusability (Inheritance and Composition)

"Inheritance is the mechanism of basing an object upon another object retaining similar implementation"[18].

"Object composition is a way to combine objects or data types into more complex ones"[19].

In UMF, if a package inherits another package, it will automatically apply all of the schemas from the inherited package. The inheritance can be achieved by "\$use". The instances will not be copied to avoid confusion, but the instances can be reused with the UMF style reference and the package-chain. This kind of reuse is a composition mode.

UMF recommends using composition in the instance segment and inheritance in the schema segment.

3.5. Language Convention

3.5.1. Full Name and Namespace

With the increase of the package number, the package names might have conflicts. UMF provides a full name system with hierarchical namespace to avoid the ambiguous package names. A full name is a package name with globally unique strings so that the package can easily be identified. URL seems to be a solution, but in this case, it can lead to redundancy. UMF provides a Java-like namespace system, which means every package serves to related group. The full name starts with a namespace of a specific group, and concatenated each hierarchy with "." . For example, the module in the script slope of Columba can be described as "columba.script.modules".

3.5.2. Access Permission

Access Permission in Instance Segment

The namespace and the top-level field together construct the hierarchy system that can distinguish the document information by packages. UMF recommends the "Access Convention" to restrict the access permission of data in different packages. Since a UMF file is a valid JSON format, it is impossible to build the compulsory access permission, but this convention can protect the packages from messy modification to some extent.

The convention for the access of data:

1. The "\$umf.common" or "\$common" data field is a field that all of the apps or services can read and write. This field can also be regarded as a "cache" for the package providers. (no reserved words)

2. Converters or develop-packages can access all of the data fields, but the write operations should be as less as possible.
3. Instance fields with a full name can only be accessed by the corresponding packages or the individual packages in 2.
4. The "output" field inside every package is a specialized field, which is readable for all other packages.

Access Permission in Schema Segment and Control Segment

The schema segment and the control segment are readable and writable by any package provider, but modification of those parts is not recommended in UMF.

3.6. Schema Sub-language

In order to achieve the "Package Model" paradigm, the sub-languages in SS are indispensable. Besides, the sub-language provides a hot update mode, which means, the maintenance of the sub-language and the main language can be separate. UMF provides two schema languages for the schema segment, namely "JSON Schema" and "OSON".

	{
	"type": "object",
{	"properties": {
"firstProp": "777@Integer"	"firstProp": {
}	"type": "integer"},
	},
	"required": ["firstProp"]
	}

Table 3.2.: left:OSON right:JSON Schema

3.6.1. JSON Schema

As mentioned in the background part, UMF must provide compatibility. JSON Schema is a universal solution to the validation of JSON format, so it should be supported in UMF. "JSON Schema is a vocabulary that allows you to annotate and validate JSON documents"[9]. A JSON Schema file is also a valid JSON format, so it can be integrated

into UMF easily. The version of JSON Schema we used here is "Draft v7", which also provides some high-order features like "contains", "if/then/else" and etc. This is a mature solution for validating the JSON format and it can almost handle every case. The completeness also makes it too complicated and redundant. Besides, the changeability is also a problem: Continuous modification and publishment with JSON Schema result in wasting time. Another issue is that the relationship between the data and the schema is too weak. The JSON Schema can accept pure syntax rule without grammar or semantic consistency. The following is the definition of Path in OpenAPI's JSON Schema, we can see the key pattern has no semantic meanings.

```
{
  "type": "object",
  "patternProperties": {
    "^\\\\": {
      "$ref": "#/definitions/PathItem"
    },
    "^x-": {
    }
  },
  "additionalProperties": false
}
```

3.6.2. OSON

OSON is a micro-language to describe the schema of JSON, inspired by "Orderly" and "Annotation". It is created and published for the main language UMF. It allows language users to use the annotation inside the existing JSON. OSON is a subset of JSON-Schema. It can be compiled to JSON-schema with the OSON compiler. Like JSON Schema, an OSON file is also a valid JSON format, so it can also be integrated into UMF easily. The version of OSON we use here is "0.9.0". The language will be updated with the development of UMF. It can not handle all of the cases as JSON Schema, but it reduces the complexity and redundancy. High-frequency changeability is a feature of OSON, which allows language users to quickly generate a validation schema and update the modification, based on an example JSON. The consistency of OSON is much better than JSON Schema. The syntax is the same structure with the grammar or semantic. It is easy to build the OSON file with an example JSON:

```
// a JSON of a book
{
  "title": "Language Design",
```

```
"price": 10,
"author": {
  "name": "Alice",
  "age": 21,
  "works": [ "Book A", "Book B", "Book C" ]
}
}
//JSON
{
  "title": "Language Design@String",
  "price": "10@Integer[1,]",
  "author": {
    "name": "Alice@String",
    "age": "21@Integer[1,]",
    "works": [ "Book A@String", "Book C" ]
  }
}
//JSON Schema
{
  "type": "object",
  "properties": {
    "title": {
      "type": "string"
    },
    "price": {
      "type": "integer",
      "minimum": 1
    },
    "author": {
      "type": "object",
      "properties": {
        "name": {"type": "string"},
        "age": {"type": "integer", "minimum": 1},
        "works": {
          "type": "array",
          "items": { "anyOf": [ {"type": "string"} ] }
        }
      }
    },
    "required": [
```

```
        "name",
        "age",
        "works"
    ]
}
},
"required": [
    "title",
    "price",
    "author"
]
}
```

Basic OSON Concept

OSON is inspired by the language "Orderly" , but from the aspect of structure and grammar, they are totally different. The idea of OSON is to finish the language inside the JSON closure so that it can be easily integrated into the UMF. Also, the OSON can be regarded as an extension or modification of JSON. The structure in JSON is reserved and used as the schema structure in OSON to keep the consistency. The syntax of OSON is a superset of JSON. In grammar, OSON inherits and modifies JSON's grammar, so the definition of OSON uses the similar form of JSON grammar.

The MF of OSON Core

OSON is a context-free language, which means the grammar of OSON can be written as a context-free grammar without "shift/reduce conflict". Since the context-free grammar of OSON is too complicated, in this article, the simplified definition of OSON will be used to explain the core concept of OSON. A BGR version of the context-free grammar of OSON will be listed in the Appendix.

```
osonSchema
    osonElement
```

```
osonElement
    osonValue
```

```
osonValue
    osonObject
    osonArray
```

annotatedValue

ojsonObject

```
'{' ojsonMembers '}'  
'{'
```

ojsonArray

```
'['  
'[' ojsonElements ']
```

ojsonMembers

```
ojsonMember ',' ojsonMembers  
ojsonMember
```

ojsonElements

```
ojsonElement ',' ojsonElements  
ojsonElement
```

ojsonMember

```
annotatedKey ':' ojsonElement
```

annotatedValue

```
DB_QUOTE ojsonValueString ojsonValueAnnotation DB_QUOTE  
JSONString
```

annotatedKey

```
DB_QUOTE ojsonKeyString ojsonKeyAnnotation DB_QUOTE  
JSONString
```

The OSON property is a pair of "Key" and "Value", denoted as "OSON Member". Both "Key" and "Value" are the type of string. So the "Annotation" in the "Key" part and the "Value" part can be modeled as two sub-problems of plain-text markup language. "ojsonValueString" and "ojsonKeyString" can be regarded as two examples of the schema, which mostly come from the example JSON of the program. "?" and "@" characters are used to solve the "control character" and the "escape character" problems in the plain-text markup languages.

OSON Annotation

Until now, the "Key" language has only one control segment, with the character "?", which represents the optional property. So, we mainly discuss the "Value" language, namely "OSON Annotation", denoted as "OSONA". OSONA will use the mathematical notation: "[]" for the range and "{}" for the set.

- Grammar of OSONA

```
annotationObject
    '@' annotation
```

```
annotation
    annotationType
```

```
annotationType
    unionType
    basicType
```

```
unionType
    '(' basicType ')' '|' unionType
    '(' basicType ')'
```

```
basicType
    basicTypePrefix basicTypeSuffix
```

```
basicTypePrefix
    INTEGER_TYPE optionalRange
    NUMBER_TYPE optionalRange
    BOOLEAN_TYPE
    NULL_TYPE
    ANY_TYPE
    STRING_TYPE optionalRange optionalPerlRegex
    JSON_TYPE
    '* optionalTypeTuple
```

```
optionalTypeTuple
    unionType
    #nothing
```



```
basicTypeSuffix
    optionalEnumValues optionalDefaultValue
```

```
optionalEnumValues
    '{ elements }'
    '{ }'
    #nothing
```

```
optionalDefaultValue
    '= json_value'
    #nothing
```

```
optionalRange
    '[' json_number ',' json_number ']'
    '[' json_number ',' ']'
    '[' ',' json_number ']'
    '[' ',' ']'
    #nothing
```

```
optionalPerlRegex
    REGEX
    #nothing
```

- **TypePrefix**

"TypePrefix" can be decomposed into two parts: "Type" and "PrefixOptional".

The purpose of the "Type" is to use the reserved keywords for expressing the types. There are six primary types in OSONA: "@Integer", "@String", "@Boolean", "@Number", "@Null", "@Any". They represent the corresponding types in JSON Schema. There are two optional types: "@JSON" and "@*" to represent the original JSON and additional properties in the value side.

The "PrefixOptional" provides the optional range for the integer and number types, while the optional regular expression for the string type. The range notation will be used for expressing the optional range, and the Perl regular expression will be used for expressing the optional regular expression. A simple example:

```
// the integer between 1 and 100
@Integer[1,100]
```

```
// the string with length 1 to 10,  
// it starts with one of those characters: "h/m/n/i/s"  
@String[1,10]/^[hmnis]/
```

- **TypeSuffix**

The suffix part is an optional part with variable length. It fills the vacancies of the expression of types and reduces the gaps between OSONA and JSON Schema. The enumeration value and default value are the two most important features. The math notation of "Set" and "Domain" is used here. A simple example:

```
// the integer between 1 and 10  
// it must be 1 to 5, the default value is 4  
@Number[1,10]{1,2,3,4,5}=4  
// prefix:  
// the string with length 1 to 10  
// the string starts with one of those characters: "h/m/n/i/s"  
// suffix:  
// it must be the element of {"hello","my","name","is","shen"}  
// the default value is "hello"  
@String[1,10]/^[hmnis]/{'hello','my','name','is','shen'}='hello'
```

- **Compile Process**

The OSONA data will be compiled to the "value part" of JSON Schema. The compiler is a state machine. A simple example:

```
@Integer compiles to {"type": "integer"} // only the value part
```

Escape

Since the "@" and "?" are used for two control characters in OSON Annotation, it is necessary to define the normal "@" and "?" characters. OSON uses "@@" and "??" in the example part for the normal characters "@" and "?". So the "osonValueString" and "osonKeyString" in the MF of OSON Core are modified strings respectively without "@" and "?". Here is an example:

```
{  
  "es??cape": "7@@7@String"  
}
```

after compiling:

```
{
  "type": "object",
  "properties": {
    "es?cape": {
      "type": "string",
      "examples": [
        "7@77"
      ]
    }
  },
  "required": [
    "es?cape"
  ]
}
```

Shift Reduce Conflict

In the conceptual grammar (see subsection 3.6.2), the "DB_QUOTE osonValueString" and "JSONString" have the Shift/Reduce (S/R) conflict. Also, "DB_QUOTE osonKeyString" and "JSONString" have the same issue. The reason is both expressions start with the double quote, which leads to "looking forward". The concept of "string" must be redefined. The S/R conflict can be then solved with the decomposition of "osonValueString" as the following syntax.

```
annotatedValue
  DBL_QUOTE closePart
```

```
closePart
  '@@' closePart
  osonAnnotation DBL_QUOTE
  singleModifiedValueChar closePart
  DBL_QUOTE
```

```
singleModifiedValueChar
  '?'
  normalChar # the characters without '?' and '@'
```

OSON to JSON Schema

The relative rules from OSON to JSON Schema are not complicated, so a non-normative example is listed here instead of a formal tutorial. Examples of OSON compared with the compiled JSON Schema:

```
{
  "firstProp": "777@Integer",
  "secondProp": "str@String{'c','a','b'}", // enum
  "thirdProp": "true@Boolean",
  "forthProp?": "optionalStr@String", // optional property
  "array1": [ "array1@String[1,10]", "ListValidation" ],
  // ListValidation has one element without annotation
  "array2": [ "arrayEle1@String", "TupleValidation@String" ]
  // TupleValidation: all elements must have annotation.
}
{
  "type": "object",
  "properties": {
    "firstProp": {"type": "integer"},
    "secondProp": {
      "type": "string",
      "enum": ["c", "a", "b"]},
    "thirdProp": {"type": "boolean"},
    "forthProp": {"type": "string"},
    "array1": {
      "type": "array",
      "items": {
        "anyOf": [
          { "type": "string",
            "minLength": 1,
            "maxLength": 10 } ] }},
    "array2": {
      "type": "array",
      "items": [
        {"type": "string"},
        {"type": "string"} ] }
  },
  "required": [
    "firstProp",
```

```
    "secondProp",  
    "thirdProp",  
    "array1",  
    "array2"]  
}
```

3.7. Conclusion

This chapter discusses the design process of UMF. UMF is a data description and schema language, but some core concepts are inspired by programming languages. These features make UMF powerful and easy to use between different programs.

4. Implementation of Compiler

As mentioned above, UMF consists of multiple sub-languages. The compiler must be decomposed into multiple parts or compilers to fit the sub-languages. A whole topology of the language partition is in Figure 4.1.

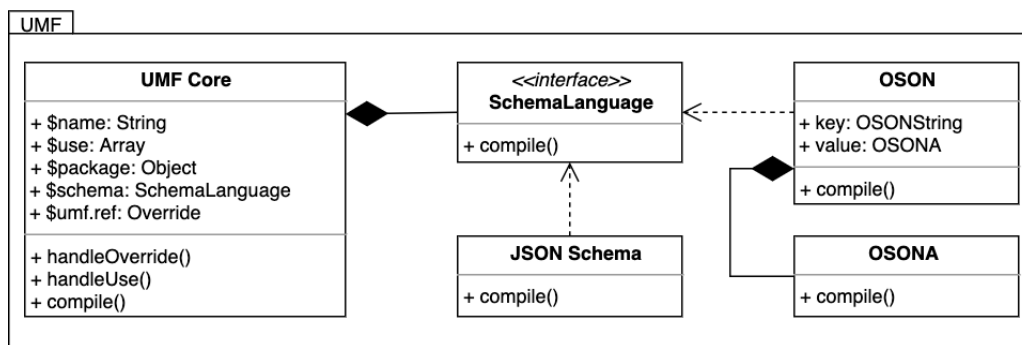


Figure 4.1.: Language Partition

4.1. Compiler: UMF Core

UMF Core is the entrance of the language. The whole compile process starts from here and will call other sub-language compilers. Since the UMF is a valid JSON format, lexical analysis and syntactic analysis can be omitted because Javascript provides a JSON parse function called "JSON.parse ()".

However, two segments in UMF Core still need to be compiled: SS and IS.

4.1.1. Compile Schema Segment

The process can be divided into 2 procedures:

1. "Desugar \$use": this procedure resolves the \$use syntax and copies the compiled results to \$package field. It may call the UMF Core's compiler recursively. The output of this procedure should be like:

```
$package: { "package1":{...}, "package2":{...} }  
$schema: { // nothing here}
```

2. "Merge schemas": this procedure starts to resolve the override and reference firstly, then compiles the \$schema field, finally merges the schemas from the \$schema field and the desugaring \$use. The output of this procedure should be like:

```
$package: { "package1":{...}, "package2":{...} }  
$schema: { "type": "object",  
           "properties":{  
             "package1":{...},  
             "package2":{...} }}}
```

In those procedures, the following methods will be invoked:

- dereference (): this method resolves the reference with UMF Path
- desugar (): this method resolves the data of UMF packages. The desugar process will delegate the resolving process to the resource center if the package data is not local.
- resourceCenter.get (): this method handles the resolving process. There are two environments: Node.js and Browser. For the browser, a JSONP protocol will be used to avoid the CORS. For the Node.js, "fs reader" or "HTTP request" will be used to read the files directly.

4.1.2. Compile Instance Segment

The only process is to resolve the override and the reference, then copy them to the corresponding location.

4.2. Automatic Lexical Analysis and Syntactic Analysis

OSON and OSONA are two plain-text markup sub-languages. The lexical analysis and the syntactic analysis can be automatic by using Flex and Bison. Flex is an automatic lexical analyzer. It is used to tokenize the input data and replace the token with the predefined symbols. GNU Bison is a parser generator that reads context-free grammar (BGR) as input and generates a parser as output[20]. The parser is a finite-state machine that is equivalent to the input context-free language. The lexical analysis with Flex can have two sections:

1. Definitions Section: Name definitions have the form: "name definition"[21]. For example,

```
ESCAPE_WS  \\["\\rnt/]
SPACE      [ ]
WSC        {ESCAPE_WS}|{SPACE}
WS         {WSC}+
```

2. Rules Section: A rule has the form: "pattern action"[22]. For example,

```
":"        return ':'
{WS}       return 'WS'
```

The syntactic analysis with Bison will be discussed together with the code generation.

4.3. Automatic Code Generation

Flex and Bison are always used together with the inline programming statements to finish the code generation process. JISON is a javascript implementation of Flex and Bison with the inline programming language Javascript[23]. Hence, JISON will be used in our project. Like Bison, Jison provides hook points in every BGR rule so that some Javascript code can be injected into the hook point.

Like the following example, the "osonValue" is defined as one of "osonObject", "osonArray" and "annotatedValue". If the "osonObject" is reduced to "osonValue" according to the grammar, the log "oson object found" will be printed.

```
osonValue
  :osonObject {console.log ("oson object found");}
 |osonArray {console.log ("oson array found");}
 |annotatedValue{console.log ("annotated value found");}
 ;
```

4.4. Compiler: OSON

The compiler of OSON is built with JISON. A whole JISON grammar will be listed in the Appendix A.1.

4.4.1. Lexical Analysis and Syntactic Analysis

In order to distinguish the normal string and the annotated string in OSON, "JSON string" is divided into two subsets:

1. The strings contain consequent odd numbers of "@" and "?", denoted as "annotated string"
2. Other strings, denoted as "normal string"

The partition of the string sets can be achieved by creating a new character set "MODIFIED_BOTH_CHAR" in the lex part, which is the JSON character set without "?", "@" and whitespace. Usually, the whitespace can be omitted outside the string, and directly shifted inside the string and this process is completed in the tokenizing process. However, there are two types of string this time. It is not possible to distinguish them in the lexical analysis. Syntactic analysis is needed, so the whitespace must be kept in lexical analysis and handled in the Bison part.

To clarify the issue, consider the following scenario:

```
"a_b_"_{}: two spaces in the word and one between the word and object
```

Flex can replace it with "STRING" and "OBJECT" if there is only one string type. The reduction process in Bison can directly use the result "STRING" and "OBJECT" without any conflict.

However, if there are two string types "annotated string" and "normal string", Flex can not decide which type of string should be used to replace here. If "annotated string" is chosen, a grammar like "A:= normal_string OBJECT" can not be reduced by Bison correctly.

4.4.2. Code Generation

OSON will be compiled to JSON Schema. Then, the validation can be finished with the validation tools, like "AJV". The code generation part is implemented by injecting the inline code with JISON. After JISON's process, a pure OSON syntax tree will be generated. The code generation will use the syntax tree and the following code templates:

```
// Only the main templates are listed here
// The OSON Value can be divided into "osonObject",
// "osonArray" and "annotatedValue".
"osonObject":
  {
```

```
        "type": "object",
        "properties":{ ... }
    }
"oSONArray":
// tuple validation or list validation (see section 3.6.2 Examples)
    {
        "type": "array",
        "items":[ ... ]
    }
or
    {
        "type": "array",
        "items":{ ... }
    }
"annotatedValue":
call OSONA compiler
```

4.5. Compiler: OSONA

The compiler of OSONA is also built with JISON. A whole JISON grammar will be listed in the Appendix A.2.

4.5.1. Lexical Analysis and Syntactic Analysis

The JSON string is defined with double-quotes. Since OSONA is inside a string of OSON, the string definition here should be different to avoid unexpected escape problem. The string in OSONA is defined with single-quotes. So the MODIFIED_UNESCAPED_CHAR and MODIFIED_ESCAPED_CHAR with this definition will be used in the lex part. Also, all of the types, whitespace, and regular expression itself should be defined in the lex part:

```
%lex
MODIFIED_UNESCAPED_CHAR    [ -&\(-\[\]-~]
MODIFIED_ESCAPED_CHAR     \\['\\bfnr/]
%%
"Integer"                  return 'INTEGER_TYPE'
"Number"                   return 'NUMBER_TYPE'
"Boolean"                  return 'BOOLEAN_TYPE'
```

```
"Null"           return 'NULL_TYPE'  
"Any"            return 'ANY_TYPE'  
"String"         return 'STRING_TYPE'  
\\/(?:[^\\/]|"\\\\"")*\\/  return 'REGEX'  
[ \\t\\n]+       /* ignore whitespace */
```

4.5.2. Code Generation

The same with OSON, code generation of OSONA will also use the syntax tree and the following code templates:

```
// Only the main templates are listed here  
"INTEGER_TYPE": {"type": "integer"}  
"BOOLEAN_TYPE": {"type": "boolean"}  
"unionType": {"type": [ ... ]}  
"NUMBER_TYPE optionalRange": {  
  "type": "number", "minimum": ..., "maximum": ...  
}  
"STRING_TYPE optionalRange optionalPerlRegex": {  
  "type": "string",  
  "minLength": ..., "maxLength": ..., "pattern": "..."  
}  
"STRING_TYPE optionalEnumValues optionalDefaultValue": {  
  "type": "string", "enum": [...], "default": "..."  
}
```

4.6. Conclusion

A compiler is the essential program of a language. This chapter discusses an implementation of the UMF's compiler based on JISON.

5. Implementation of IDE

This chapter assesses the Integrated Development Environment (IDE) of UMF. The IDE is based on the Monaco Editor. "The Monaco Editor is the code editor that powers VS Code"[24].

5.1. Dynamic Packages Loading

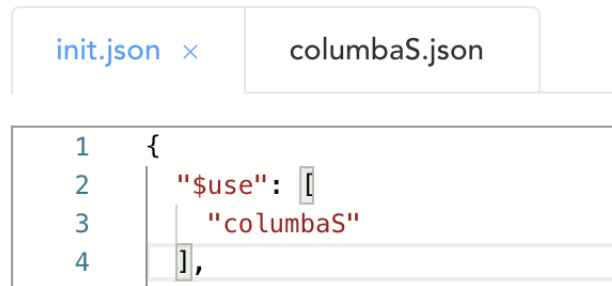
UMF allows users to import packages with "\$use" syntax. After the user types the package name in the \$use field, the package should be immediately loaded, and the schema and instance from the package should be loaded into the IDE environment for dynamic checking. However, during the typing, the whole UMF file is not a valid JSON format, so the compiler can not be used directly. "String Matching" methods will mismatch the cases if some JSON strings also contain the "\$use" substring, so a small language can be created to solve this problem. The IDE needs to parse the language with the following grammar to identify the "\$use" field, then invokes the desugaring of "\$use".

```
useObject:= "$use" ':' '[' packageDeclarationArray ']'
packageDeclarationArray:= packageDeclaration ',' packageDeclarationArray
                           |packageDeclaration
packageDeclaration:=     STRING_LIT
                           |packageUse
```

An example view of the dynamic package loading is in Figure 5.1.

5.2. Dynamic Syntax Generation and Validation


After pulling the package data by dynamic loading, the IDE should merge the package schema with the UMF Core schema which is preloaded in the IDE. The \$schema field also has the same problem as the \$use field. A small dynamic language of \$schema is also to be created. After compiling the \$schema field, this part of the schemas also



```
1 {
2   "$use": [
3     "columbaS"
4   ],
```

Figure 5.1.: Dynamic Package Loading

needs to be merged. Monaco Editor will set this schema as the validation option to validate the UMF file. An example view is in Figure 5.2



```
1 {
2   "$use": [
3     "columbaS"
4   ],
5   "columbaS": {
6     "columba.rules": {
7       "control_channel_width": "123",
8       "control_inlet_dimension": 1500,
9       "flow_channel_width": 100,
10      "inlet_dis": 2000,
11      "min_spacing": 100
12    }
13  }
14 }
```

init 1 of 1 problem

Incorrect type. Expected "integer".

Figure 5.2.: Dynamic Validation

5.3. Debounce and Stable Strategy

The process mentioned before will be executed every time the user inputs a new character. It may cause busy waiting and slow feedback. Debounce technology can solve this problem. Debounce is a front-end web technology to control the execution times inside a time slot. It is implemented by setting a timeout event to invoke the delegated function. If this function is called again inside a period, the countdown of the timeout event will be reset to ensure that the function is only executed once. Besides, the IDE also adopts a stable strategy, which means if the dynamic field is not modified, the process will not be executed. This stable strategy is achieved by using the cache.

5.4. Virtual File System in Browser

Monaco Editor provides the view and data based on the model (MVC-pattern). The model is a data structure that contains file information. It uses URI to distinguish the files. A URI consists of "scheme", "authority" and "path". For example, the URI of Columba's module can be written as "umf://columba/module". The scheme is "umf", the authority is "columba" and the path is "/module". The IDE can swap files by loading different models. At the runtime, the IDE keeps all models in the storage, saves the user's modification. There are four APIs that the Monaco Editor provides for models: "getModels", "setModel", "saveViewState", and "restoreState". An example view is in Figure 5.3



Figure 5.3.: Virtual File System

5.5. Auto-complement

The modern IDE usually provides the auto-complement. "Autocomplete, or word completion, is a feature in which an application predicts the rest of a word a user is typing." Monaco Editor provides a "languages" module that allows registering the customized language, so the auto-completion can be implemented by defining a new language based on JSON. An example view is in Figure 5.4.

5.6. Foldable JSON View

Since the language is JSON, the IDE provides the "fold" function to fold the JSON data. An example view is in Figure 5.5.

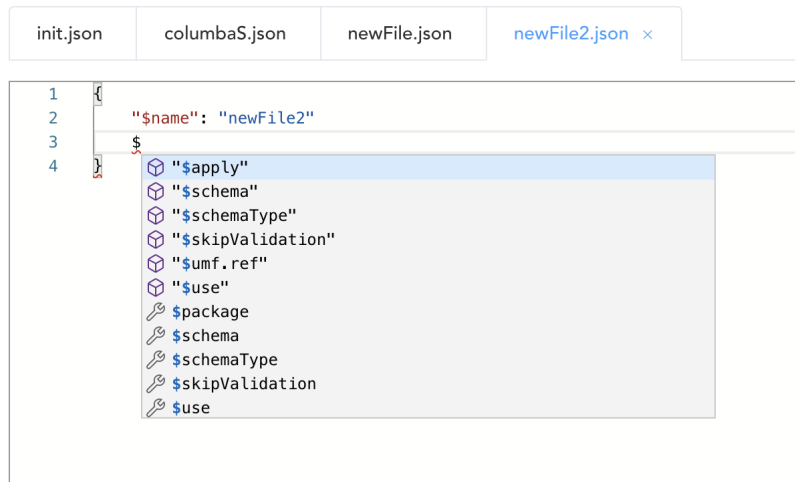


Figure 5.4.: Auto-complement

5.7. Monaco Package

Dynamic loading and validation is the stateful process corresponding to a specific model, so models should keep the state information: dynamic package and diagnostic information. The IDE encapsulates a file model with the state information into a "Monaco Package". It makes the swapping procedure easy.

5.8. Conclusion

The IDE provides some convenient functions to read and edit the UMF file. This chapter discusses an implementation of the UMF's IDE based on Monaco Editor.

5. Implementation of IDE

```
1  {
2    "$use": [
3      "columbaS"
4    ],
5    "columbaS": {
6      "columba.rules": {
7        "control_channel_width": "123",
8        "control_inlet_dimension": 1500,
9        "flow_channel_width": 100,
10       "inlet_dis": 2000,
11       "min_spacing": 100
12     },
13     "columba.script": {
14 >   "columba.modules": {...
15     },
16     "columba.netList": [],
17     "columba.parallel": [],
18     "columba.conflict": []
19   }
20 }
21 }
```

Figure 5.5.: Foldable JSON View

6. Applications

6.1. Integrating UMF into "Cloud-Columba"

6.1.1. The Class Diagram of Columba S

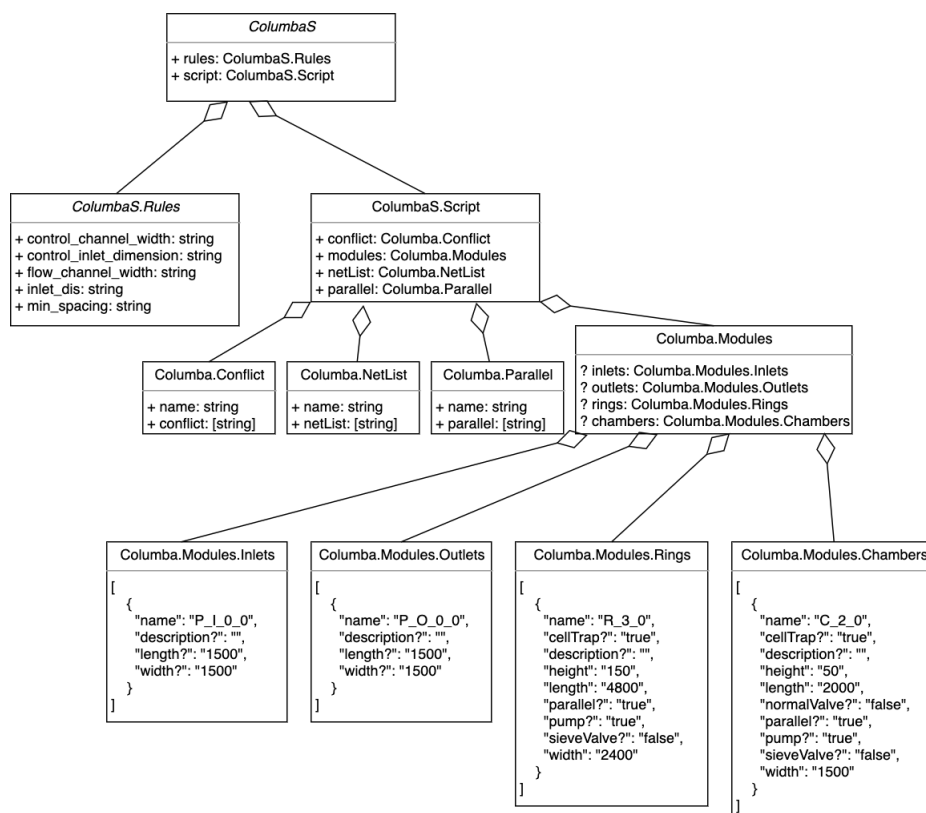


Figure 6.1.: The Class Diagram of Columba S

6.1.2. Build Package "ColumbaS"

As the data structure in Figure 6.1, the "ColumbaS" package can be built from bottom to top. UMF can quickly implement the four basic modules. An example of the package "inlets" is here:

```
{
  "$name": "columba.modules.inlets",
  "$schemaType": "OSON",
  "$schema": [
    {
      "name": "P_I_0_0@String='P_I_0_0'",
      "description?": "@String='this is a Inlet'",
      "length?": "1500@Integer='1500'",
      "width?": "1500@Integer='1500'"
    },
    "array"
  ]
}
```

To aggregate the four modules together, a "modules" package is needed here:

```
{
  "$name": "columba.modules",
  "$use": [
    "columba.modules.inlets",
    "columba.modules.outlets",
    "columba.modules.chambers",
    "columba.modules.rings"
  ]
}
```

After building the "modules" package, the other script parts, namely "conflict", "netList", and "parallel" should be achieved then. They are very similar, the "netList" package is listed here:

```
{
  "$name": "columba.netList",
  "$schemaType": "OSON",
  "$schema": [
    {
      "from": "name@String",

```

```
    "to": "name@String"
  },
  "array"
]
}
```

Also, the "script" package is added to aggregate the four script parts. It is similar to the "modules" package. Then a full "columbaS" package can be finished by using the "script" and "rules" package:

```
{
  "$name": "columbaS",
  "$use": ["columbaS.rules", "columbaS.script"],
  "$schema": {
    "$apply": true
  }
}
```

6.1.3. JMESPath converter

The Cloud-Columba provides two modes for the input: UMF and Simple. The "UMF" mode is an online IDE: the users write the code down. The "Simple" mode is an online form: the users need to follow the guild to finish the form. The form data can also be regarded as a JSON format, so a JSON data converter is required here to switching the input mode.

Since Columba S is still developing, the data format may be changed in the future, so a general method is chosen: JMESPath. JMESPath, as a query language for JSON, provides the power of reshaping the JSON data[25]. Here is an example of reshaping the data structure "netList":

```
// It will filter the outlet module, and rebuild a new JSON
modules[?type!='p_o'].instances[].{name:name,netList:netList}
```

6.1.4. Current View

The IDE of UMF is integrated into Cloud-Columba, Figure 6.2 shows the current view.

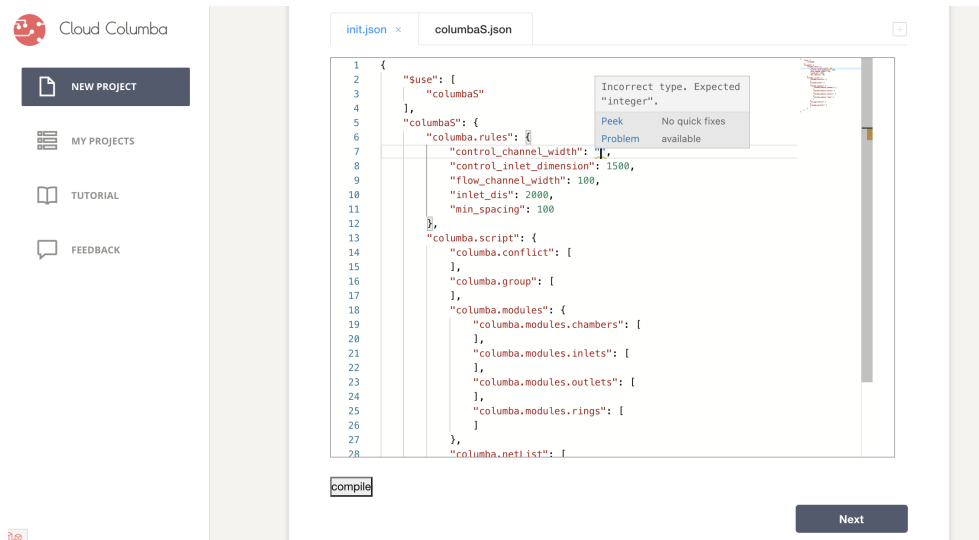


Figure 6.2.: The Current View of the IDE in Cloud-Columba

6.2. "Cloud-Columba" with "Parchmint" Benchmark

6.2.1. Conversion

Parchmint contains the low level information, while Columba S uses the high level information. Some data in the Parchmint file can be directly referred into Columba S, for example the "name" property:

```
{
  "$use": [
    "columbaS",
    {
      "$packageName": "hiv1_p24_immunoassay",
      "$url": "https://.../hiv1_p24_immunoassay.json"
    }
  ],
  "...": {
    // after compiling: "name": "Source1"
    "name": {
      "$umf.ref": "hiv1_p24_immunoassay/components/0/name"
    }
  }
}
```

Other data needs to be added manually to fill the gaps, here is a compiled version of the Parchmint's example "hiv1_p24_immunoassay":

```
"columba.modules.rings": [  
  {  
    "name": "Mixer1",  
    "cellTrap": true,  
    "description": "",  
    "height": 150,  
    "length": 4800,  
    "parallel": true,  
    "pump": true,  
    "sieveValve": false,  
    "width": 2400  
  }  
],  
"columba.modules.inlets": [  
  {  
    "name": "Source_1",  
    "description": "",  
    "length": 1500,  
    "width": 1500  
  },  
  ...  
]  
....  
"columba.netList": [  
  {  
    "from": "Source1",  
    "to": "Mixer1"  
  },  
  ...  
  {  
    "from": "Source4",  
    "to": "Mixer1"  
  },  
  {  
    "from": "Mixer1",  
    "to": "Trap1"  
  }  
]
```

```

    }
    ...
]

```

Figure 6.3 and Figure 6.4 show the output of Columba S and the visualization of the original benchmark.

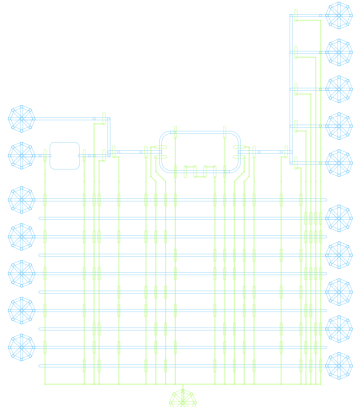


Figure 6.3.: The output of Columba S

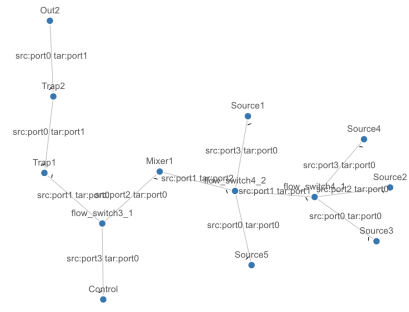


Figure 6.4.: The Parchmint's Visualization of the original Benchmark

6.2.2. Converter

A formal converter is a better solution for the conversion of different format. The whole process should be like Figure 6.5: the converter converts the Parchmint entries to UMF of the "ColumbaS" package, and Columba S runs the task and saves the output to the "ColumbaS" package, then the results can be compared. Furthermore, the converter can also convert the output of Columba S back to the Parchmint's format and Parchmint can then visualize it with simple graph like Figure 6.4.

6.3. Workflow Platform "Cloud-Columba"

UMF standardizes the interfaces of different projects with packages. A new project can directly use the formats of the dependent projects or create the converters based on the packages of dependent projects. For example, VOM can directly use the output of Columba S, while Parchmint can implement a converter to exchange the information with Columba S.

"Cloud-Columba" can become a workflow platform in the future with the data storage

6. Applications

of UMF format. In Figure 6.5, all of the user inputs will be described with UMF to enter the Cloud-Columba platform. All of the intermediate programs will use UMF to process the data and communication. The output of the Cloud-Columba platform is also a UMF format, which can be finally rendered by the front-end visualization program via Javascript. The result can be texts, pictures or even interactive animations.

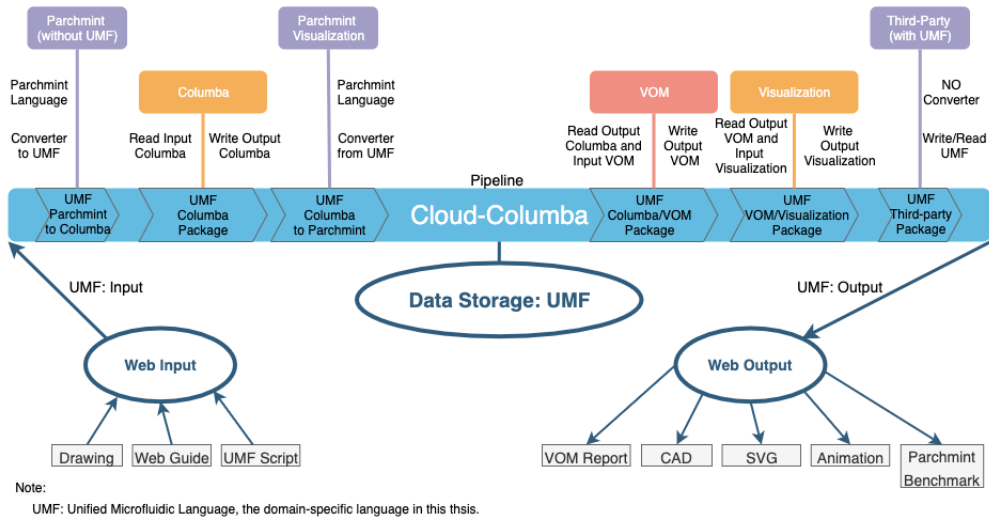


Figure 6.5.: The pipeline platform "Cloud-Columba"

7. Conclusion

The domain-specific language UMF is developed to standardize the description of data in the automated design of microfluidic biochips. Users can quickly generate the input and output format as well as the validation schema with UMF. The UMF's compiler and IDE provide a flexible development environment for the language users. In the future, UMF may become a standard data description language in the Cloud-Columba platform, and it can be extended easily to help the researchers in this field.

List of Figures

1.1. The Plain-text Input	1
1.2. Programs Topology	2
3.1. The Class Diagram of Columba S	7
3.2. \$Package Field	12
4.1. Language Partition	30
5.1. Dynamic Package Loading	37
5.2. Dynamic Validation	37
5.3. Virtual File System	38
5.4. Auto-complement	39
5.5. Foldable JSON View	40
6.1. The Class Diagram of Columba S	41
6.2. The Current View of the IDE in Cloud-Columba	44
6.3. The output of Columba S	46
6.4. The Parchmint's Visualization of the original Benchmark	46
6.5. The pipeline platform "Cloud-Columba"	47

List of Tables

3.1. JSON Pointer	17
3.2. OSON and JSON Schema	19

Bibliography

- [1] T.-M. Tseng, M. Li, D. N. Freitas, A. Mongersun, I. E. Araci, T.-Y. Ho, and U. Schlichtmann. "Columba S: A scalable co-layout design automation tool for microfluidic large-scale integration." In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.
- [2] M. Li, T.-M. Tseng, Y. Ma, T.-Y. Ho, and U. Schlichtmann. "VOM: Flow-Path Validation and Control-Sequence Optimization for Multilayered Continuous-Flow Microfluidic Biochips." In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2019, pp. 1–8.
- [3] J. Huang. "Visualization of the Fluid Behavior on Microfluidic Large-Scale Integration Biochips." In: (2019).
- [4] B. Crites, R. Sanka, J. Lippai, J. McDaniel, P. Brisk, and D. Densmore. "ParchMint: A Microfluidics Benchmark Suite." In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, pp. 78–79.
- [5] T.-M. Tseng, M. Li, Y. Zhang, T.-Y. Ho, and U. Schlichtmann. "Cloud Columba: Accessible Design Automation Platform for Production and Inspiration." In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016.
- [6] A. Van Deursen and P. Klint. "Domain-specific language design requires feature descriptions." In: *Journal of computing and information technology* 10.1 (2002), pp. 1–17.
- [7] B. DuCharme. *XML: The annotated specification*. Prentice Hall PTR, 1998.
- [8] O. Ben-Kiki, C. Evans, and B. Ingerson. *Yaml specification*. 2005.
- [9] <https://www.json.org/json-en.html>.
- [10] Wikipedia contributors. *Backus–Naur form — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Backus%E2%80%93Naur_form&oldid=955734338. [Online; accessed 9-June-2020]. 2020.
- [11] <https://www.crockford.com/mckeeman.html>.
- [12] A. K. Lee, W.-H. Ip, and K.-L. Yung. "Inheritance and polymorphism in real-time monitoring and control systems." In: *Journal of Intelligent Manufacturing* 11.3 (2000), pp. 285–294.

- [13] O. Initiative et al. "OpenAPI specification." In: URL: <https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1> (2014).
- [14] R. Buyya, S. T. Selvi, and X. Chu. *Object-oriented programming with Java: essentials and applications*. Tata McGraw-Hill, 2009.
- [15] Wikipedia contributors. *Dependency grammar* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Dependency_grammar&oldid=950473663. [Online; accessed 9-June-2020]. 2020.
- [16] *rfc6901*. <https://tools.ietf.org/html/rfc6901>. [Online; accessed 9-June-2020].
- [17] M. Droettboom et al. "Understanding JSON Schema." In: Available on: <http://spacetelescope.github.io/understanding-jsonschema/UnderstandingJSONSchema.pdf> (accessed on 14 April 2014) (2015).
- [18] Wikipedia contributors. *Inheritance (object-oriented programming)* — *Wikipedia, The Free Encyclopedia*. [https://en.wikipedia.org/w/index.php?title=Inheritance_\(object-oriented_programming\)&oldid=956864028](https://en.wikipedia.org/w/index.php?title=Inheritance_(object-oriented_programming)&oldid=956864028). [Online; accessed 9-June-2020]. 2020.
- [19] Wikipedia contributors. *Object composition* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Object_composition&oldid=959104894. [Online; accessed 9-June-2020]. 2020.
- [20] *GNU Bison*. <https://www.gnu.org/software/bison/>. [Online; accessed 9-June-2020].
- [21] *GNU Bison Definitions Section*. <https://www.cs.virginia.edu/~cr4bd/flex-manual/Definitions-Section.html#Definitions-Section>. [Online; accessed 9-June-2020].
- [22] *GNU Bison Rules Section*. <https://www.cs.virginia.edu/~cr4bd/flex-manual/Rules-Section.html#Rules-Section>. [Online; accessed 9-June-2020].
- [23] *JISON*. <https://zaa.ch/jison/>. [Online; accessed 9-June-2020].
- [24] *Monaco Editor*. <https://microsoft.github.io/monaco-editor/>. [Online; accessed 9-June-2020].
- [25] R. Queirós. "SeCoGen-A Service Code Generator." In: *8th Symposium on Languages, Applications and Technologies (SLATE 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

A. Appendix

A.1. OSON Grammar

```
%lex
KEY_ANNOTATION_SYMBOL      [\?]
VALUE_ANNOTATION_SYMBOL    [@]
DIGIT1to9                  [1-9]
DIGIT                      [0-9]
DIGITS                     (?:{DIGIT}+)
INT
    (?:{DIGIT1to9}{DIGITS}|{DIGIT}|"-"{DIGIT1to9}{DIGITS}|"-"{DIGIT})
FRAC                       (?:\.{DIGITS})
EXP                       (?:{E}{DIGITS})
E                          [eE] [+ -]?
HEX_DIGIT                  [0-9a-f]
UNESCAPED_CHAR_WITHOUT_AT [ -!#-\?A-\[\]-~]
UNESCAPED_CHAR_WITHOUT_QM [ -!#->@\[\]-~]
UNESCAPED_CHAR_WITHOUT_BOTH [ -!#->A-\[\]-~]
UNESCAPED_CHAR_WITHOUT_BOTH_WS [!#->A-\[\]-~]
ESCAPEDCHAR                \\["\\bfnrt/]
ESCAPEDCHAR_WITHOUT_WS    \\["\\bf/]
UNICODECHAR
    \\u{HEX_DIGIT}{HEX_DIGIT}{HEX_DIGIT}{HEX_DIGIT}
UNESCAPEDCHAR              [ -!#-\[\]-~]
CHAR                       {UNESCAPEDCHAR}|{ESCAPEDCHAR}|{UNICODECHAR}
CHARS                      {CHAR}+
MODIFIED_VALUE_CHAR
    {UNESCAPED_CHAR_WITHOUT_AT}|{ESCAPEDCHAR}|{UNICODECHAR}
MODIFIED_KEY_CHAR
    {UNESCAPED_CHAR_WITHOUT_QM}|{ESCAPEDCHAR}|{UNICODECHAR}
MODIFIED_BOTH_CHAR
```

A. Appendix

```
{UNESCAPED_CHAR_WITHOUT_BOTH_WS}|{ESCAPEDCHAR_WITHOUT_WS}|{UNICODECHAR}
DBL_QUOTE          ["]
ESCAPE_WS          \\["\\rnt/]
SPACE              [ ]
WSC                {ESCAPE_WS}|{SPACE}
WS                 {WSC}+
```

```
%%
{VALUE_ANNOTATION_SYMBOL}          return '@';
{KEY_ANNOTATION_SYMBOL}            return '?';
{VALUE_ANNOTATION_SYMBOL}{VALUE_ANNOTATION_SYMBOL}  return '@@'
{KEY_ANNOTATION_SYMBOL}{KEY_ANNOTATION_SYMBOL}      return '??'
"{                                return '{'
"}                                return '}'
"[                                return '['
"]                                return ']'
",                                return ','
":                                return ':'
{WS}                              return 'WS'
{MODIFIED_BOTH_CHAR}              return 'MODIFIED_BOTH_CHAR';
{DBL_QUOTE}                        return 'DBL_QUOTE'
\\(?: [^\\/] | "\\|/")*\\/         return 'REGEX'
[ \\t\\n]+                          /* ignore whitespace */
/lex
```

```
%start osonSchema
```

```
%%
```

```
osonSchema
```

```
    :osonElement
```

```
    ;
```

```
osonElement
```

```
    : WS osonValue WS
```

```
    | WS osonValue
```

```
    | osonValue WS
```

```
    | osonValue
```

```
    ;
```

```
osonValue
```

```
:osonObject
|osonArray
|annotatedValue
;
osonObject
: '{ WS }'
| '{ }'
| '{ osonMembers }'
;
osonArray
: '[ WS ]'
| '[ ]'
| '[' osonElements ]'
;
annotatedValue
: DBL_QUOTE closePart
;
closePart
: '@' '@' closePart
| '@' osonAnnotation DBL_QUOTE
| singleModifiedValueChar closePart
| DBL_QUOTE
;
singleModifiedValueChar
: '?'
| normalChar
;
osonAnnotation
: singleModifiedValueChar normalChars
;
normalChars
: normalChar normalChars
| '@' normalChars
| '?' normalChars
|
;
normalChar
: MODIFIED_BOTH_CHAR
| '{'
```

```
|'}'  
|'['  
|']'  
|','  
|':'  
|WS  
;  
osonMembers  
  :osonMember ',' osonMembers  
  |osonMember  
  ;  
osonElements  
  :osonElement ',' osonElements  
  |osonElement  
  ;  
osonMember  
  :annotatedKey ':' osonElement  
  |WS annotatedKey WS ':' osonElement  
  |WS annotatedKey ':' osonElement  
  |annotatedKey WS ':' osonElement  
  ;  
annotatedKey  
  :DBL_QUOTE closeKey  
  ;  
closeKey  
  :DBL_QUOTE  
  |singleModifiedKeyChar closeKey  
  |'?' '?' closeKey  
  |'?' DBL_QUOTE  
  ;  
singleModifiedKeyChar  
  : '@'  
  |normalChar  
  ;
```

A.2. OSONA Grammar

```
%lex
```


A. Appendix

```
DIGIT1to9          [1-9]
DIGIT              [0-9]
DIGITS            (?:{DIGIT}+)
INT
    (?:{DIGIT1to9}{DIGITS}|{DIGIT}|"-"{DIGIT1to9}{DIGITS}|"-"{DIGIT})
FRAC              (?:\.{DIGITS})
EXP              (?:{E}{DIGITS})
E                [eE] [+ -]?
HEX_DIGIT        [0-9a-f]
NUMBER
    (?:{INT}{FRAC}{EXP}|{INT}{EXP}|{INT}{FRAC}|{INT})
MODIFIED_UNESCAPED_CHAR  [ -&\(-\[\]-~]
MODIFIED_ESCAPED_CHAR   \\['\\bfnrt/]
UNICODE_CHAR
    \\u{HEX_DIGIT}{HEX_DIGIT}{HEX_DIGIT}{HEX_DIGIT}
MODIFIED_CHAR
    {MODIFIED_UNESCAPED_CHAR}|{MODIFIED_ESCAPED_CHAR}|{UNICODE_CHAR}
MODIFIED_CHARS        {MODIFIED_CHAR}+
QUOTE                 [']

%%
{QUOTE}{QUOTE}|{QUOTE}{MODIFIED_CHARS}{QUOTE}      return 'STRING_LIT'
{NUMBER}                                             return 'NUMBER_LIT'
"true"                                              return 'TRUE'
"false"                                             return 'FALSE'
>null"                                             return 'NULL'
"Integer"                                          return 'INTEGER_TYPE'
"Number"                                          return 'NUMBER_TYPE'
"Boolean"                                         return 'BOOLEAN_TYPE'
"Null"                                           return 'NULL_TYPE'
"Any"                                             return 'ANY_TYPE'
"String"                                          return 'STRING_TYPE'
"JSON"                                           return 'JSON_TYPE'
"{"                                              return '{'
"}"                                              return '}'
"="                                              return '='
"["                                              return '['
"]"                                              return ']'
```

A. Appendix

```
"("          return '('
")"          return ')'
"|"          return '|'
"'"          return "'"
","          return ','
":"          return ':'
"*"          return '*'
"@"          return '@'
\\(?:[^\\/]|"\\\/")*\\
[ \t\n]+     /* ignore whitespace */
/lex
```

```
%start annotationObject
%%
```

```
annotationObject
    : '@' annotation
    ;
annotation
    : annotationType
    ;
annotationType
    : unionType
    | basicType
    ;
unionType
    : '(' basicType ')' '|' unionType
    | '(' basicType ')'
    ;
basicType
    : basicTypePrefix basicTypeSuffix
    ;
basicTypePrefix
    : INTEGER_TYPE optionalRange
    | NUMBER_TYPE optionalRange
    | BOOLEAN_TYPE
    | NULL_TYPE
    | ANY_TYPE
```

```
|STRING_TYPE optionalRange optionalPerlRegex
|JSON_TYPE
|'*' optionalTypeTuple
;
optionalTypeTuple
: unionType
|
;
basicTypeSuffix
:optionalEnumValues optionalDefaultValue
;
optionalEnumValues
:'{' elements '}'
|'{ '}'
|
;
optionalDefaultValue
: '=' json_value
|
;
optionalRange
:[' json_number ',' json_number ']'
|[' json_number ',' ']'
|[' ',' json_number ']'
|[' ',' ']'
|
;
optionalPerlRegex
:REGEX
|
;
elements
:json_value
|json_value ',' elements
;
json_value
:json_string
|json_number
|TRUE
```

```
|FALSE
|NULL
;
json_number
: NUMBER_LIT
;
json_array
: '[' ' ']'
| '[' elements ']'
;
members
: pair
| pair ',' members
;
pair
: STRING_LIT ':' json_value
;
json_string
: STRING_LIT
;
```